REMARKS

Applicants appreciate the Examiner's notification that the Information Disclosure Statement filed on May 4, 2005, contained an illegible document. A legible copy of the reference has been enclosed so that the reference may be fully considered.

Claims 1-10, 13 and 17-101 are pending in the application. Claims 1-10, 13, and 17-101 have been rejected. Claims * and * have been amended. No new matter has been added.

Rejection of Claims under 35 U.S.C. § 101

Claims 22-26, 36-38 and 46-58 stand rejected under 35 U.S.C. § 101 because the claimed invention is directed to non-statutory subject matter. Applicants respectfully disagree with the rejection but have amended the independent claims 22, 46, and 54 to show the structural and functional interrelationships between the data structure and other claimed aspects of the invention which permit the data structure's functionality to be realized.

The Office Action further states that claims 22-26, 36-38, and 46-58 are non-statutory because they are "not tangibly embodied. Claims 22 and 46 recite 'a computer readable medium' ... and the specification discloses the computer-readable medium as including transmission media such as digital and analog communication link[s].... Transmission signals are incapable of being touched or perceived absent the tangible medium through which they are conveyed." (See Office Action dated May 27, 2005, paragraph 6, pages 3-4). Applicants respectfully traverse this rejection.

Applicants note that independent claims 22, 46, and 54 have been amended to include functional descriptive material in the form of instructions. As noted in the Examination Guidelines for Computer-Related Inventions, "[w]hen functional descriptive material is recorded on some computer-readable medium it becomes structurally and functionally interrelated to the medium and *will be statutory* in most cases." MPEP §2106, page 2100-12, emphasis added. Furthermore, "[c]laims that recite nothing but the physical characteristics of a form of energy, such as a frequency, voltage, or the strength of a magnetic field, define energy or magnetism, per se, and as such are nonstatutory natural phenomena... However, a *signal claim directed to a practical application of electromagnetic energy is statutory* regardless of its transitory nature." MPEP §2106, page 2100-14, citing O'Reilly v. Morse, 56 U.S. (15 How) 62, 114-119 (1853) and

In re Breslow, 616 F.2d 516, 519-21, 205 USPQ 221, 225-26 (CCPA 1980), emphasis added. Independent claims 22, 46, and 54 are clearly directed to more than simply the physical characteristics of a form of energy, as indicated by the description of the instructions that have been encoded on the computer-readable medium. Furthermore, a data signal embodied in a carrier wave including instructions for accessing database tables is a practical application of electromagnetic energy due to the encoding of the instructions, which include functional descriptive material, on the carrier wave. Accordingly, independent claims 22, 46, and 54 recite statutory subject matter.

Consequently, independent claim 22, its dependent claims 23-26 and 36-38, independent claim 46, its dependent claims 47-53, independent claim 54, and its dependent claims 55-58 are allowable.

Rejection of Claims under 35 U.S.C. § 102

Wagner reference

Claims 22-26, 36-38 and 46-58 stand rejected under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent 6,092,102 ("Wagner"). Applicants respectfully traverse this rejection.

Amended claim 46 includes the following limitation:

a database comprising:
a user interface object table comprising information regarding a user interface object of a user interface to communicate with a communication channel;

Independent claims 54 and 22 have been amended to include substantially the same limitation.

The Office Action states that a user interface object table is taught by database 138 and 140, which are shown in Fig. 3 and column 11, lines 31-39 of Wagner. (See Office Action dated May 27, 2005, paragraph 9, pages 4-5.) However, database 138 is described as "an N-dimensional database defining the communication channel(s) (CHANNEL) based on the user (RECIPIENT), the message type (MT) and other arguments ARGs (e.g., time of day, role of the user, setting of the patient within the hospital enterprise, relationship of the clinician or user to the patient." (See Wagner, column 11, lines 43-49.) Database 140 is described as storing preferences of the users for the characteristics of the communication (e.g., reliability, time latency). (See Wagner, column 11, lines 31-37.) Neither of these databases includes a table or

any information related to a "user interface object of a user interface to communicate with a communication channel."

The Office Action points out the notifier front end 122, which "provides a suitable user interface (GUI) to I/O device 128, such as a display and a keyboard and/or mouse, that permits a particular user, after security login, to modify the data in the preferences 34 and functions 36." (See Office Action dated May 27, 2005, paragraph 9, pages 4-5.) Applicants acknowledge that the database described in Wagner can be modified by a user via a user interface. However, this capability does not imply that the database itself contains information related to a user interface for communicating via a communication channel.

The user interface of Wagner is described as being used to modify preferences 34 and functions 36. Preferences 34 are described as "a mapping of message types to preferred communication channels ... for each of the users," information which is unrelated to a user interface for communication via the communication channel. (See Wagner, column 6, lines 45-47.) Functions 36 are described as being "employed to select one or more of the users ... to receive the message of the data structure ... over one or more of the communication channels... . The functions 36 include, among others, a mapping of message types with respect to some or all of the users ... ." Similarly, this information is unrelated to a user interface for communication via the communication channel.

Applicants submit that the user interfaces used for communication via the communication channels of Wagner are likely to be channel-specific user interfaces unrelated to the GUI described with respect to the notifier front end 122. For example, if an e-mail message is to be sent to a user, the notifier function 6 communicates "a suitably formatted user message 38 to the appropriate user(s) by employing the appropriate communication channel(s)." (See Wagner, column 6, lines 59-65.) The user then likely uses an e-mail client user interface that is specific to his e-mail service to access the message 38.

Applicants can find no teaching in Wagner of a user interface object table for a user interface used to communicate via a communication channel. Because all limitations of independent claims 46, 54, and 22 are not taught by Wagner, independent claim 46 and its dependent claims 47-53, independent claim 54 and its dependent claims 55-58, and independent claim 22 and its dependent claims 23-26 and 36-38 are allowable for at least this reason.

Rejection of Claims under 35 U.S.C. § 102

Rahman reference

Claims 1-10, 13, 17-21, 34, 39-42, 59-63, 67-76 and 84-94 stand rejected under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent 6,463,292 ("Rahman"). Applicants respectfully traverse this rejection.

Two limitations of amended independent claim 1 are shown below:

> identifying a channel driver comprising a command associated with the activation of the work item object; and
> causing the channel driver to issue the command to an outgoing communication channel of the communication channels.

Each of independent claims 1, 13, 17, 19, 21, 39, 59, 67, 84, and 85 has been amended to include substantially similar limitations.

Rahman does not teach "identifying a channel driver comprising a command associated with the activation of the work item object; and causing the channel driver to issue the command to an outgoing communication channel of the communication channels." Applicants have searched Rahman and can find no description of a channel driver that issues commands to communication channels and no description of identifying such a channel driver associated with activation of the work item object. Because all limitations of the independent claims are not taught by Rahman, independent claim 1, its dependent claims 2-10 and 27-33, independent claim 13, its dependent claims 95-101, independent claim 17, its dependent claims 18, 34, and 35, independent claim 19, its dependent claim 20, independent claim 21, independent claim 39, its dependent claims 40-45, independent claim 59, its dependent claims 60-66, independent claim 67, its dependent claims 68-83, independent claim 84, independent claim 85, and its dependent claims 86-94 are allowable.

Rejection of Claims under 35 U.S.C. § 103

Claims 27-33, 35, 43-45, 77-83 and 95-101 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent 6,463,292 ("Rahman") in view of U.S. Patent 6,389,132 ("Price"). Applicants respectfully traverse this rejection.

Each of claims 27-33, 35, 43, 77-83, and 95-101 depends from one of independent claims 1, 13, 17, 39, or 67. Each of independent claims 1, 13, 17, 39, and 67 has been shown to be allowable over the Rahman reference standing alone. Consequently, each of claims 27-33, 35, 43, 77-83, and 95-101 is allowable for at least the foregoing reasons.

Rejection of Claims under 35 U.S.C. § 103

Claims 64-66 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent 6,463,292 ("Rahman") in view of U.S. Patent 6,092,102 ("Wagner"). Each of claims 64-66 depends from independent claim 59, which has shown to be allowable over the Rahman reference standing alone. Consequently, claims 64-66 are allowable for at least the foregoing reasons.

CONCLUSION

In view of the amendments and remarks set forth herein, the application is believed to be in condition for allowance and a notice to that effect is solicited. Nonetheless, should any issues remain that might be subject to resolution through a telephonic interview, the Examiner is invited to telephone the undersigned at 512-439-5086.

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail in an envelope addressed to: Mail Stop AF, COMMISSIONER FOR PATENTS, P. O. Box 1450, Alexandria, VA 22313-1450, on July 27, 2005.

Attorney for Applicant(s) 7/27/05
Date of Signature

Respectfully submitted,

D'Ann Naylor Rifai
Attorney for Applicants
Reg. No. 47,026
(512) 439-5086 [Phone]
(512) 439-5099 [Fax]

XP 000488600

PUBLICATION DATE: 28. 11. 94
(further bibliographic data on next page)

Experiences with an Object-Oriented Architecture for Developing Dynamically Extensible Distributed System Management Software

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130
(314) 935-6160

Tatsuya Suda
suda@ics.uci.edu
Information and Computer Science Department
University of California, Irvine, CA 92717
(714) 856-4105[1]

p. 500 – 506 = (7)

Abstract

Developing extensible, robust, and efficient distributed applications is a complex task. In order to help alleviate this complexity, we have developed the ADAPTIVE Service eXecutive (ASX) framework. ASX is an object-oriented framework composed of automated tools and reusable C++ components that simplify the development, configuration, and reconfiguration of distributed applications in a heterogeneous environment. These applications may be configured dynamically to contain multiple network services that execute concurrently in one or more processes or threads. Components in the ASX framework have been ported to UNIX and Windows NT and are currently being used in a number of large-scale production distributed systems. This paper describes our experience gained using the ASX framework to build a highly modular, reusable, and extensible software architecture for a family of distributed system management applications.

1 Introduction

The demand for extensible, robust, and efficient distributed systems is increasing dramatically in research and commercial environments. Although distributing application services among a set of autonomous hosts offers many potential benefits, developing distributed systems is more complex than developing non-distributed systems. A significant portion of this complexity arises from limitations with conventional tools and design techniques used to develop distributed application software. In particular, network programming tools (such as sockets, named pipes, and RPC) available in contemporary operating systems (such as UNIX, Windows NT, and OS/2) lack type-safe, portable, re-entrant, and extensible programming interfaces. For example, both sockets and named pipes identify endpoints of communication via weakly-typed I/O descriptors. The use of these descriptors increases the potential for subtle run-time errors. Another major source of complexity stems from the widespread use of algorithmic design techniques to develop distributed application software [1]. Many distributed systems are developed using algorithmic design techniques that result in monolithic, non-extensible software architectures [2].

Object-oriented frameworks help to alleviate the complexity associated with developing distributed application software. A framework is an integrated collection of software components that collaborate to produce a reusable architecture for a family of related applications [3]. Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration of complex applications in domains such as graphical user interfaces [4], databases [5], operating system kernels [6], and communication subsystems [7, 2].

The components in a framework typically include *classes* (such as message managers and timer-based event managers, and connection maps [8]), *class hierarchies* (such as an inheritance lattice of mechanisms for local and remote interprocess communication [9]), *class categories* (such as a family of concurrency mechanisms [2]), and *objects* (such as an event demultiplexer [10]). By emphasizing the integration and collaboration of application-specific and application-independent components, frameworks enable larger-scale reuse of software, compared to reusing individual classes and stand-alone functions.

To illustrate how object-oriented frameworks are being applied successfully in practice, this paper examines the features, structure, and usage of the ADAPTIVE Service eXecutive (ASX). We developed the ASX framework to provide an integrated collection of reusable, object-oriented network software components. These components simplify the development of distributed applications by enhancing the modularity, extensibility, reusability, and portability of software that utilizes operating system (OS) concurrency, explicit dynamic linking, interprocess communication (IPC), and I/O demultiplexing mechanisms.

In addition to describing the object-oriented architecture of the ASX framework, this paper describes our experiences using the ASX framework to develop commercial software for a family of distributed system management applications at a major telecommunications company. These applications manage private-branch exchange (PBXs) switches and public central-office telecommunication switches across hetero-

[1] This research is supported in part by grants from the University of California MICRO program, Hughes Aircraft, Nippon Steel Information and Communication Systems Inc. (ENICOM), Hitachi Ltd., Hitachi America, Tokyo Electric Power Company, and Hewlett Packard (HP).

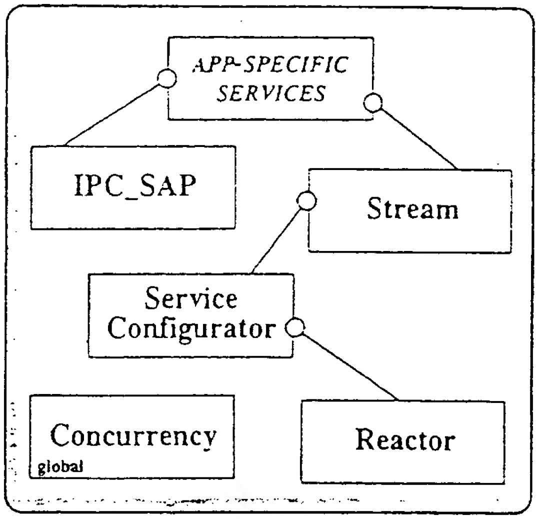

Figure 1: Class Categories in the ASX Framework

geneous hardware and software platforms.

This paper is organized in the following manner: Section 2 outlines the architectural components in the ASX framework: Section 3 examines the object-oriented structure of a production distributed Call Center Management system built using ASX; and Section 4 presents concluding remarks.

2 The Object-Oriented Architecture of the ASX Framework

This section outlines the major class categories in the ASX framework. A class category is a collection of software components that collaborate to provide a set of related services [1]. The architecture of the ASX framework was developed incrementally by generalizing from extensive experience with building a range of distributed systems (including on-line transaction processing systems [10], telecommunication switch monitoring systems [11], and parallel communication subsystems [2]). After developing several prototypes and iterating through a number of alternative designs, the class categories illustrated in Figure 1 were identified and implemented.[2]

Using ASX a distributed application may be formed by combining and customizing components in each of the following class categories via object-oriented language features such as data abstraction, inheritance, dynamic binding, object composition, and template instantiation:

• **Stream Class Category:** These components are responsible for coordinating the configuration and run-time execution of one or more Streams [2]. A Stream is an object used to configure and execute application-specific services in the ASX framework. A Stream contains a series of interconnected `Module` objects that may be linked together *statically* by developers at compile-time or *dynamically* by administrators or applications at installation-time and at run-time. `Modules` are used to decompose the architecture of a distributed application into functionally distinct layers. Each layer implements a cluster of related services (such as an end-to-end transport service, a presentation layer formatting service, or an event routing service for monitoring the behavior of telecom switches). Every `Module` contains a pair of `Queue` objects that are used to partition a layer into its constituent read-side and write-side processing functionality.

A distributed application may be implemented as an interconnected series of `Module` objects that communicate by exchanging typed message objects with adjacent `Modules`. `Modules` may be joined together statically and/or dynamically in essentially arbitrary configurations in order to satisfy application requirements and enhance component reuse.

• **Service Configurator Class Category:** These components are responsible for inserting and removing the run-time address bindings of services implemented by `Modules` in shared object libraries. The `Service Configurator` components provide an extensible object-oriented interface and a configuration scripting language that automates the use of OS mechanisms for explicit dynamic linking [11]. This enables one or more Streams to be dynamically reconfigured *without* requiring the modification, recompilation, relinking, or restarting of executing applications.

• **Reactor Class Category:** These components are responsible for demultiplexing I/O-based events received on communication ports, time-based events generated by a timer-driven callout queue, or signal-based events [10]. When these events occur at run-time, the `Reactor` dispatches the appropriate pre-registered handler(s) to process the events. The `Reactor` encapsulates the `select`, `poll`, and `WaitForMultipleObjects` I/O demultiplexing system calls via a portable, extensible, and type-safe object-oriented interface. `Select` and `poll` are UNIX system calls that detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously. `WaitForMultipleObjects` is a Windows NT system call that provides similar demultiplexing functionality.

• **Concurrency Class Category:** These components are responsible for spawning, executing, synchronizing, and gracefully terminating services at run-time via one or more threads of control [2]. In the ASX framework, services may be executed at run-time using several different types of process and thread mechanisms provided by the underlying OS. By decoupling service behavior from the type of mechanisms used to invoke a service, the ASX framework increases the range of concurrency configuration alternatives available to developers [2].

[2] Throughout the paper, object-oriented component relationships are illustrated via Booch notation [1]. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate a link exists between two objects. Solid rectangles indicate class categories, which combine a number of related classes into a common name space.

• IPC_SAP Class Category: These components are responsible for receiving and transmitting data with participating communication peers residing on processes in local or remote hosts. The IPC_SAP components standard OS local and remote IPC mechanisms (such as UNIX sockets and Windows NT named pipes) within a type-safe object-oriented interface [9]. To improve service portability, the IPC_SAP classes may be used in conjunction with object-oriented language features (such as inheritance and parameterized types) to minimize an application's reliance on a particular type of IPC mechanism.

The lines that connect the class categories in Figure 1 indicate dependency relationships. For example, components implementing the application-specific services in a particular distributed applications depend on the Stream components, which in turn depend on the Service Configurator components. Since components in the Concurrency class category are used throughout the application-specific and application-independent portions of the ASX framework, they are marked with the global adornment.

The ASX framework incorporates concepts from several other modular communication frameworks such as the System V STREAMS [12], the x-kernel [8], and the Conduit framework [7] from the Choices object-oriented operating system. These frameworks all contain features that support the flexible configuration of network software via the interconnection of building-block protocol and service components. In general, these frameworks encourage the development of standard communication-related components by decoupling application-specific processing functionality from the application-independent communication framework infrastructure.

3 Implementing a Distributed System with the ASX Framework

The ASX framework is being used to develop a family of distributed system management applications that monitor and control PBX and central-office telecommunication switches in a Call Center Management (CCM) system. A CCM system provides a set of services that allow the staff of a call center (such as an airline reservation center or an insurance claims processing center) to assess the call center performance and the quality of service provided to customers. This section describes the behavior of the CCM system and illustrates ASX framework components that were used to implement the distributed CCM software.

3.1 Overview of the Call Center Management System

The ASX-based CCM system controls the processing of information that is generated continuously by system operators, telecommunication switches, and networks. Supervi-

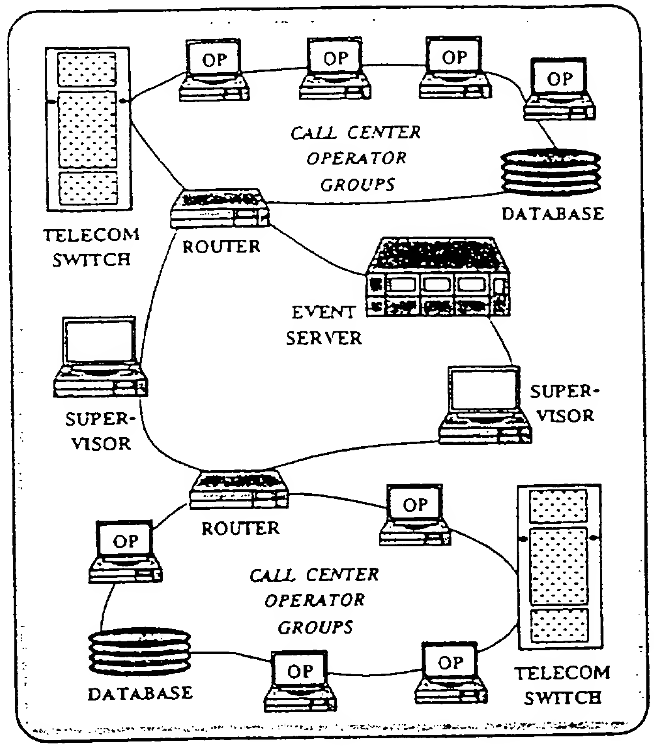

Figure 2: Distributed Components in the Call Center Management System

sors use this information to interactively monitor and optimize system performance, as well as to forecast future allocation of resources (such as operators and switch capacity) to meet customer demands.

Figure 2 illustrates the distributed architecture of the CCM system. In this system, telecommunication switches route incoming customer calls to an operator. Operator interactions with customers are expedited by graphical user interfaces on hosts that access a database of customer account records. The CCM system continuously generates performance data (known as "activity events") that reports the activities of operators and switches. These activity events are sent automatically to a central event server, which runs on a separate network connected to the operator group networks. The event server is a mediator that analyzes, filters, and forwards the activity events it receives to other hosts throughout the network. These hosts summarize the activity events they receive, and display them to supervisors in a concise, graphical format.

The object-oriented design and implementation of the CCM system is strongly influenced by requirements for platform independence and configuration flexibility. Platform independence is necessary since the CCM system is targeted for various configurations of telecommunication switches (such as PBX and central-office switches), host platforms (such as Windows NT, Windows 3.1, OS/2, and UNIX), and wide-area and local-area networks (such as X.25, TCP/IP,

and Novell IPX/SPX). Configuration flexibility is necessary since not all call center installation sites require every feature provided by the CCM system. It would be possible (although highly undesirable) to manually construct and deliver one or more CCM systems that are customized for the platforms and the subsets of features required by a particular site. However, such a static configuration process would require that the selection of services and the division of labor between different hosts in a distributed CCM system be completely fixed during initial system deployment. Our experience with earlier-generation CCM systems indicated that even if this information was available at the time of deployment, it was likely to change in the future, often upon short notice.

The ASX framework facilitates both platform independence and configuration flexibility to improve software component reuse across platforms and to reduce development effort. For example, C++ abstract base classes, inheritance, dynamic binding, and parameterized types are used extensively throughout the CCM system to localize and minimize platform dependencies. Likewise, the ASX framework is used to defer the point of time at which a particular set of services are configured together to form a CCM application. By combining advanced OS features (such as multi-threading and explicit dynamic linking) and C++ language features (such as templates, inheritance, and dynamic binding), the ASX framework enables services offered by CCM applications to be extended without modifying, recompiling, relinking, or even restarting the system at run-time [11].

3.2 Mapping CCM Functionality onto ASX Components

Figure 3 illustrates the ASX framework components used to implement the event server portion of the CCM system (the other components in the CCM system are not described in this paper). The event server performs the following tasks:

- It receives activity events generated continuously by operator hosts and telecom switches
- It analyzes and filters the activity events it receives to determine which actions to perform and which supervisors should receive which incoming activity events
- It forwards the filtered activity events across a network to the subset of supervisor hosts that have previously registered to receive these events

The CCM event server is composed of four hierarchically-related `Modules` that may be configured statically and/or dynamically by the run-time environment available in the ASX framework. The use of ASX `Modules` helps to improve the platform independence of the CCM system by encapsulating non-portable system mechanisms (such as communication protocols and activity event frame formats) behind abstract interfaces. This section describes the behavior of the `Switch_Adapter`, `Event_Analyzer`, `Event_Filter`, and `Session_Router` `Module` objects that comprise the CCM event server.

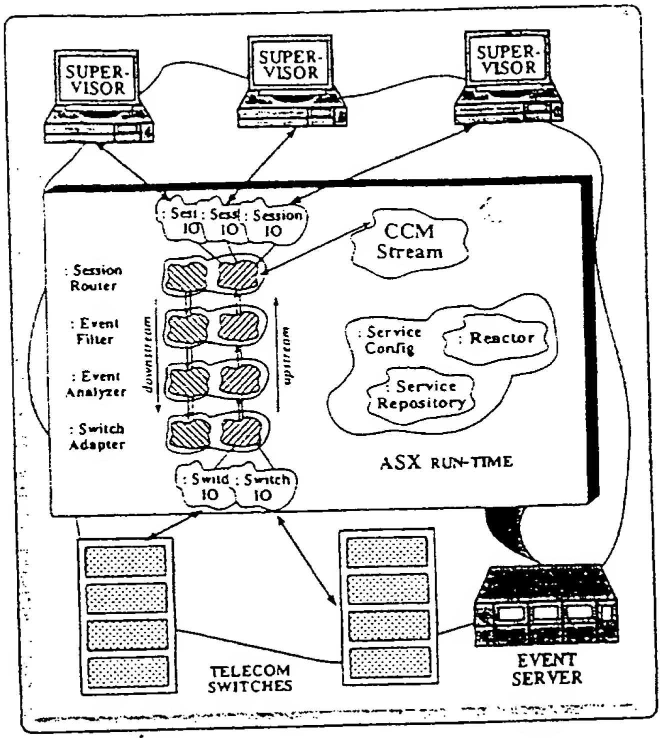

Figure 3: ASX Components in a Distributed Call Center Manager

3.2.1 The Switch_Adapter Module

The `Switch_Adapter Module` object coordinates communication with telecom switches monitored by the CCM event server. This `Module` shields the higher layers of the event server architecture from switch-specific communication characteristics (such as activity event frame formats). The `Switch_Adapter Module` maintains a collection of `Switch_IO` objects that are responsible for parsing incoming activity events. These activity events are transformed and encapsulated into a canonical switch-independent message object, which is built atop a flexible message management class described in [2]. After being allocated and initialized, the incoming canonical message objects are passed upstream to the `Event_Analyzer Module`.

3.2.2 The Event_Analyzer Module

The `Event_Analyzer Module` is used to transform switch-specific activity events into a switch-independent format. This transformation process helps to improve system portability. For instance, only the `Event_Analyzer` portion of the event server was affected significantly when the original PBX-based version of the CCM system was ported to a different central-office switch architecture. During the event analysis process, the `Event_Analyzer` also synthesizes switch-independent derived events that are triggered off the occurrence of one or more switch-specific events. After the `Event_Analyzer` has transformed and/or synthesized

incoming activity events, it forwards the new events to the Event_Filter Module.

3.2.3 The Event_Filter Module

The Event_Filter Module minimizes unnecessary network traffic by forwarding only those activity events that at least one supervisor has registered to receive. The Event_Filter contains a collection of Event Forwarding Discriminator (EFD) objects. An EFD object contains a predicate that indicates the type of activity event(s) a supervisor wants to receive. An EFD predicate may be used to selectively filter out activity events based on criteria such as event type, event value, event generation time, and event frequency. An EFD predicate may contain relational operators that allow the composition of arbitrarily complex filter expressions.

During system configuration, supervisors may register EFD objects with the event server. During system execution, the event server inspects the registered EFDs to determine the set of supervisors that should receive each incoming activity event. If an activity event matches a supervisor's EFD predicate, the supervisor's addressing information is added to the Session_Set in the message object that encapsulates the activity event. After all the EFDs are inspected by the Event_Filter Module, the message object containing the activity event and the Session_Set is passed upstream to the Session_Router Module.

3.2.4 The Session_Router Module

The Session_Router Module is a reusable ASX component. It shields the lower layers of the CCM event server from non-portable details of the communication protocols used to communicate with supervisors. Supervisors connect to the event server by establishing a session with the Session_Router Module. A separate Session_IO object is created to manage each supervisor session. This Session_IO object handles all the data transfer and control operations between the event server and a supervisor. After connecting to the event server, a supervisor indicates the type of activity event(s) he or she would like to monitor. Subsequently, when the Session_Router receives a message object from the Event_Filter Module, it automatically multicasts the message to all the supervisors indicated by the addressing information residing in the message object's Session_Set.

The Service_Config object illustrated in the middle of Figure 3 is a reusable component from the ASX framework's Service Configurator class category [11]. The event server uses this object to control the initial configuration, subsequent reconfiguration(s), and termination of Modules from the Stream class category. Modules may be configured statically at installation-time or reconfigured dynamically at run-time. The Service_Config object integrates other ASX framework components such as the Service_Repository and the Reactor. The Service_Repository is an object manager that simplifies the run-time configuration and administration of the Modules used to implement protocol stacks in a Stream. The Reactor is an event demultiplexer that dispatches incoming messages from supervisors and activity events from switches to the appropriate Session_IO and Switch_IO event handlers, respectively. Messages arriving from supervisors are received by a Session_IO object, sent downstream through the CCM_Stream object, and handled by the appropriate Module (*e.g.*, EFD registrations are handled by the Event_Filter Module). Likewise, incoming events from switches are received by a Switch_IO object and sent upstream starting at the Switch_Adapter Module.

3.3 CCM Event Server Configuration

The Modules that comprise the CCM_Stream object may be configured into the event server at installation-time by developers, as well as at run-time by system administrators or by applications. The ASX framework provides this degree of flexibility by combining OS explicit dynamic linking mechanisms with a configuration scripting language (described in [11]. For example, the Service Configurator class category uses following configuration script to determine which services to dynamically link into the address space of the event server at installation-time:

```
stream CCM_Stream dynamic
  STREAM * /svcs/CCM_Stream.so:alloc() {
  dynamic Switch_Adapter
    Module * /svcs/SA.so:alloc()  "-p 2001"
  dynamic Event_Analyzer
    Module * /svcs/EA.so:alloc()
  dynamic Event_Filter
    Module * /svcs/EF.so:alloc()
  dynamic Session_Router
    Module * /svcs/SR.so:alloc()  "-p 2010"
}
```

The configuration script shown above indicates the order in which the four Modules in the CCM event server are dynamically linked and pushed onto the CCM_Stream object. During the installation of the event server, the Service_Config class parses this configuration script and carries out the directives described by each entry, as follows:

1. The dynamic directive instructs the ASX framework to dynamically link the shared object file (specified by a pathname ending in .so) into the address space of the CCM event server
2. The framework the extracts and invokes the dynamically linked alloc function, which allocates an instance of the specified Module object from the shared object library
3. The framework then invokes the init methods on the two Queues in the Module, passing in any initialization parameters (which may appear as string literals at the end of the line)

4. At this point, the framework enters an event loop that waits for control messages to arrive from supervisors or for activity events to arrive from switches and/or operator hosts

When events arrive at run-time, the `Reactor` automatically dispatches the appropriate methods of the `Switch_IO` and `Session_IO` objects to initiate Stream processing.

3.4 CCM Event Server Reconfiguration

This section motivates and illustrates the dynamic reconfiguration mechanisms provided by the ASX framework. A major objective of the CCM project was to allow developers to decide very late in the development cycle (*i.e.*, at installation-time or run-time) which services would run in supervisor hosts and which would run in the event server. Our experience with earlier versions of the CCM system indicated that it was difficult to determine the appropriate mapping of services onto hosts *a priori* since processing characteristics, workloads, and OS/hardware platforms vary over time.

The run-time control environment provided by the ASX framework is used to support the CCM flexible reconfiguration requirements. This flexibility proved to be quite useful for the CCM project since different OS/hardware platforms and different network characteristics required different service configurations. For example, in some environments the event server performed most of the work since it ran on a multi-processor platform, whereas the supervisor hosts were inexpensive PCs attached to the event server via low-bandwidth networks. Conversely, in other environments the supervisor hosts performed most of the work since these hosts were powerful workstations connected to a high-speed network.

The `CCM_Stream` shown in Figure 3 performs all event analysis and event filtering processing directly in the event server. However, this configuration may not be appropriate for certain CCM environments. For example, performance may be degraded if supervisors configure a large number of Event Forwarding Discriminator (EFD) objects into an event server. In this case, the centralized event server may become a bottleneck and performance would suffer, even if there were surplus processing capacity available in the network and in the supervisor hosts. Figure 4 illustrates how the configuration shown in Figure 3 may be modified to operate efficiently in a distributed environment where event server processing constitutes the primary performance bottleneck.

The following script may be used to dynamically reconfigure `Modules` in the CCM system:

```
suspend CCM_Stream
stream CCM_Stream {
  remove Event_Filter
}
remote "-h all -p 911" {
  stream CCM_Stream {
    dynamic Event_Filter
      Module * /svcs/EF.so:alloc()
  }
}
resume CCM_Stream
```

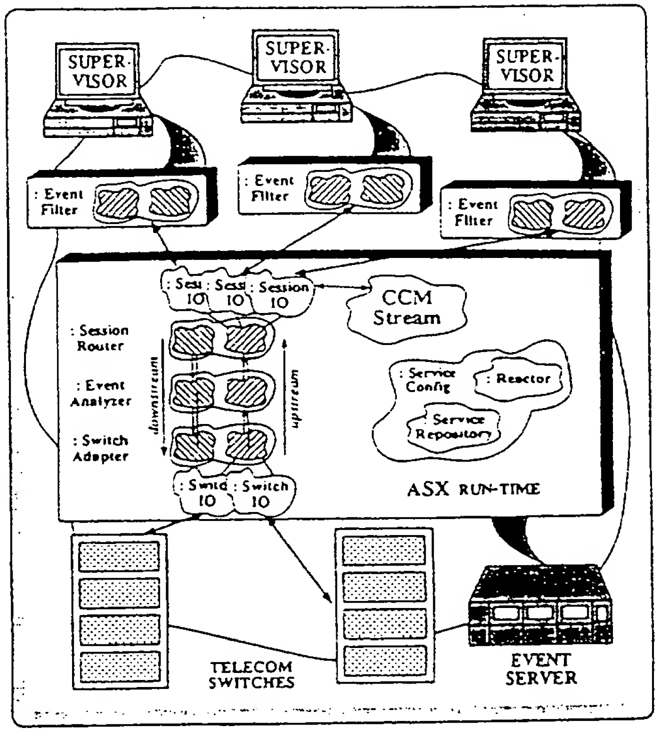

Figure 4: Reconfiguration of the Call Center Manager

This new script transfers the event filtering functionality from the event server to the supervisor hosts using the following steps:

1. It suspends the event server's `CCM_Stream` object
2. It removes the `Event_Filter Module` from the `CCM_Stream` and dynamically unlinks the associated shared object library
3. It then dynamically links the `Event_Filter Module` into Streams on all the supervisor hosts
4. Finally, it resumes the event server's `CCM_Stream` object

As a result of this reconfiguration, the overhead of event filtering is distributed among all the supervisor hosts, rather than being centralized at the event server.

We are currently evaluating the performance of the CCM distributed architecture depicted in Figures 3 and 4 to determine how to parallelize the CCM event server more effectively. Using the ASX framework, it is straightforward to reconfigure the binding of threads onto `Modules` or messages in order to reduce programming effort and improve performance [2]. We are also investigating service migration policies to formulate guidelines that ensure the dynamic reconfiguration of the event server does not disrupt or corrupt active services. A more ambitious long-term project

involves using the ASX reconfiguration mechanisms to experiment with service migration policies that relocate certain services dynamically to reduce overall system workload at run-time.

4 Concluding Remarks

The ASX framework provides an object-oriented infrastructure that supports static and/or dynamic configuration of network services that execute within one or more OS processes and threads. The object-oriented design principles underlying the ASX framework separate policies from mechanisms via object-oriented language features (such as abstract base classes, inheritance, dynamic binding, and parameterized types). This separation of concerns enhances the reuse of common distributed application software components. Component reuse is also facilitated in the ASX framework by decoupling the higher-level application-specific policies (such as activity event filtering) from the lower-level application-independent mechanisms (such as the choice of mechanisms for network communication, event demultiplexing and service dispatching, and process/thread execution agents).

In addition, the ASX framework helps to decouple application-specific service functionality from the binding onto OS processes and threads in order to improve flexibility and performance. Explicit dynamic linking and dynamic binding are also utilized to help improve extensibility and permit fine-grained time/space tradeoffs. Together, the object-oriented design principles and OS/language features facilitate the development of network services that may be updated and extended without modifying, recompiling, relinking, or restarting existing applications at run-time [11].

The ASX framework described in this paper has been used in a production environment to simplify the configuration, installation, and administration of a family of distributed system management applications for a major telecommunications company. By using the ASX framework, developers have been able to enhance distributed application functionality and reliability, as well as fine-tune system performance, without extensive redevelopment and re-installation effort. For example, debugging a faulty service typically involves dynamically reinstalling a functionally equivalent service containing additional instrumentation that helps to isolate the source of the erroneous behavior.

Thus far, the primary obstacles encountered by using object-oriented techniques and C++ have been primarily managerial and tool-related, rather than technical problems. For example, it is difficult to find experienced systems analysts, designers, and programmers who are intimately familiar with applying C++ and object-oriented design methods in distributed communication system environments. Furthermore, the level of maturity of many C++ compilers and language processing tools has been inadequate on UNIX, Windows NT, and OS/2 platforms. For example, many C++ debuggers do not support multi-threading correctly and many C++ compilers implement only a subset of the language.

Over time, it is likely that the tool-related concerns will become less problematic, particularly once the ISO/ANSI C++ standard is adopted. However, for the interim period, it is essential to staff complex software projects carefully to minimize development risks. For the CCM project, we found it useful to hire a small number of experienced OOD/C++ experts, who have worked closely with less experienced developers to shepard them through the object-oriented learning curve.

Components in the ASX framework are being used in a number of large-scale distributed systems including the AT&T Q.port ATM signaling software product, the network management portion of the Motorola IRIDIUM global personal communications system, and a family of telecommunication switch management systems developed at Ericsson/GE mobile communications. Public domain versions of the ASX framework described in this paper is available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`.

References

[1] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[2] D. C. Schmidt and T. Suda, "Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance," in *Proceedings of the 4^{th} International Workshop on Protocols for High-Speed Networks*, (Vancouver, British Columbia), IFIP, August 1994.

[3] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[4] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *USENIX C++ Workshop November*, November 1987.

[5] D. Batory and S. W. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, Oct. 1991.

[6] R. Campbell, V. Russo, and G. Johnson, "The Design of a Multiprocessor Operating System," in *USENIX C++ Conference Proceedings*, pp. 109–126, USENIX Association, November 1987.

[7] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *USENIX C++ Conference Proceedings*, pp. 191–203, USENIX Association, April 1990.

[8] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[9] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[10] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *1^{st} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), ACM, August 1994.

[11] D. C. Schmidt and T. Suda, "The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons," in *The Proceedings of the Second International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 190–201, IEEE, Mar. 1994.

[12] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.